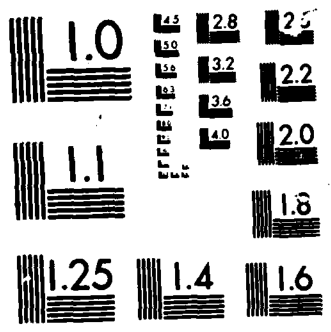
AD-A166 603 THE USE OF VALUES WITHOUT NAMES IN A PROGRAMMING SUPPORT ENVIRONMENT(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) M STANLEY NOV 85 1/1

UNCLASSIFIED RSRE-MEMO-3901 DRIC-BR-98723 F/G 9/2 NL

END


MICROCOP' CHART

UNLIMITED

BR98723

2

AD-A166 603

RSRE
MEMORANDUM No. 3901

ROYAL SIGNALS & RADAR ESTABLISHMENT

THE USE OF VALUES WITHOUT NAMES IN A PROGRAMMING SUPPORT ENVIRONMENT

Author: M Stanley

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

86 4 15 224

RSRE MEMORANDUM No. 3901

UNLIMITED

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 3901

TITLE: THE USE OF VALUES WITHOUT NAMES IN A PROGRAMMING SUPPORT ENVIRONMENT

AUTHOR: MARGARET STANLEY

DATE: NOVEMBER 1985

SUMMARY

The Flex PSE, developed at RSRE, Malvern, allows objects to be used without requiring that they be given names. This paper discusses how this is achieved, and the benefit to the user.

The use of values without names in a programming support environment

CONTENTS

The use of values without names in a programming support environment

1. Introduction

This paper discusses the effect on the user of a Programming Support Environment (PSE) in which it is unnecessary to give names to objects in order to keep them or use them. The PSE is a highly interactive environment called Flex[5], that is perceptibly different from other PSEs. It is built on the Flex capability object oriented computer architecture[1,2] developed at RSRE, Malvern.

Names on Flex do not have the importance that they assume on most other PSEs because values need not be named before they can be held on filestore, used and located. Values can be accessed by scanning a structured filestore without the need to use names to identify them. The ability to use values without naming them is shown to be not only possible but also beneficial.

2. What is FLEX ?

Flex is a multi-language Programming Support Environment (PSE) designed to simplify the development and maintenance of complex software, with a high regard for system integrity and reliability. The PSE development since the first Flex architecture came into use in 1978 has been mainly a response to requests from programmers using the system. The large software base includes all normal operating system facilities and many other procedures including compilers for Algol68 and Pascal. An Ada(*) compiler is near completion and an ML compiler is under development.

The Flex capability computer architecture has (so far) been implemented in microcode on four hardware configurations, the most recent being the ICL Perq. The implementation with which I am most familiar is a multi-user system in which 3 Flex computers share a common filestore and common peripherals.

A full description of Flex is beyond the scope of this paper, which will concentrate on those aspects that relate to the use of values and names with some discussion of the effect this has on programmers and on the method of use of the Flex PSE.

*Ada is a registered trademark of the US DoD.

3. Names and values

3.1 Values

On conventional systems values on filestore (often called files) are usually accessed through dictionaries or directories which are associations of names with files. Each user has one or more private directories of name/file associations which give him access to the files that he owns. He usually has the right to update such a directory and the values to which it gives access. Every file has at least one name.

On Flex individual filestore values do not have to be named. A filestore value on Flex can be a structure containing other filestore values (see discussion on edfiles, below) and filestore values can exist within a structure independent of any name. Only the outermost structure needs a name.

Every object or value handled on Flex, whether on filestore or in mainstore, has an associated Flex mode[4] (value type) that is used to indicate how the value is to be interpreted, and the operations that are valid on it. Values vary in type from simple integers to much more complicated structures. In contrast to the limited set of file types available in most computer systems the variety of Flex modes is boundless and most mainstore objects (including procedures) have direct analogues in Flex filestore. Users can define new Flex modes to describe new kinds of value. Whereas naming conventions are employed on some systems to indicate file types such as source text, compiled code and program, names are not needed on Flex to indicate the mode of a value. The mode of a Flex value can always be displayed on request and correct use of values of any mode is checked by the command interpreter.

In general, filestore values on Flex cannot be modified: they can only be created or used. Filestore values, once created, are constants. The only exceptions are a few root pointers which are updated as a single atomic action. The updatable root pointers are single word blocks usually containing a reference to a dictionary from which other filestore values can be reached. The architecture ensures that the root pointer update is unitary. Even dictionaries are not updated. The only way to change a dictionary is to create a new one (which is an edited version of the old dictionary) and then update the root pointer to point to the new dictionary.

3.2 Capability values

Those values in Flex which require some degree of access control to preserve system integrity are represented by capabilities. To achieve the necessary access control the Flex microcode allows access to such a value only through the capability for the value. A capability can be created and modified only by the Flex microcode although capabilities can be treated like other values in that they can be held on filestore, used in programs and can be passed to another user, giving the other user access to the controlled value. There are mainstore capabilities that control access to mainstore objects (such as procedures), filestore capabilities that control access to objects on filestore and remote capabilities that control access to remote facilities (e.g. objects on other Flex computers).

In a sense a capability is a pointer created on behalf of the user by the microcode but the capability also contains information on the type of use (read only; read/write; execute) that will be permitted by the microcode. The mode of a capability value indicates the mode of the value to which it gives access.

On conventional systems a file may be copied to more than one user. Each user will allocate a name in his directory to his copy of the file, and the names may be different. On Flex filestore values are never copied. Each user of a filestore value needs the appropriate capability, although only one copy exists of the value to which the capability gives access. Like other values, capabilities do not need names, so the potential confusion of associating the same filestore value with different names in different directories can be avoided.

3.3 Procedure values

In conventional operating systems a procedure is placed in a context by forming it into program before it can be invoked from the command interpreter. On Flex a procedure is a context independent value, like any other value. The value of a procedure is a capability to execute the procedure. A procedure can be invoked directly by the command interpreter (curt)[3] (itself a procedure) by including in the command either the unnamed procedure value or (if the procedure value has been named) the name of the procedure value. Of course a procedure is defined and invoked by name in the source text of a programming language, and that name is preserved in the external specification of the compiled unit in which it is defined, so that it can be invoked from other separately compiled units. However, the source text name has no meaning at the command language level. Like other values on Flex, procedure values do not necessarily have names at the command language level. If a name is used in a command, curt first converts the name to its associated value.

A procedure called by curt takes a value holding the procedure parameters and delivers the result as an unnamed value which may be re-used immediately in a new command or retained for future use. The Flex mode of a procedure value indicates the modes of both the parameter and the result and curt will call a procedure only on values of the correct mode.

3.4 Structured Edfiles

As mentioned above, unnamed filestore values can be held in structures on filestore. One such structure is a filestore value of Flex mode Edfile, a value that can be processed by the Flex screen editor. A user normally interacts with Flex using the Flex screen editor, which, as well as editing text, can manipulate values of any mode and can call procedures via the command interpreter (curt). Thus an editable file (an edfile) is a database-like object that may contain a mixture of characters, other edfiles, integers, filed procedures (i.e. a value held on filestore from which a procedure value can be loaded) etc. Since any edfile may contain others, edfiles may be a structure of edfiles and other values, with the restriction (enforced by the Flex architecture) that the structure must be acyclic.

The following is a trivial example of the content of a structured edfile as displayed by the Flex editor. The unnamed values are displayed in cartouches (boxes). Anything not enclosed in a cartouche is normal text.

This is the first line of the outermost level of this edfile.
This line includes an Integer 35, value 35.

Note that the value is held anonymously. The string displayed in the cartouche is not a name: it is a label selected by the user to describe the value. The label is completely independent of the value. It is associated only with a cartouche representing the value, and can be changed without affecting the value. On request the editor will replace the displayed label with a label indicating the mode of the value.
The value 35 could also be displayed as Int .

The unnamed (labelled) edfile ed_date_doc (listed below) contains a procedure and its associated documentation.

Next we display a command, which can be obeyed directly from the editor. It calls the algol68 compiler on the source edfile:

source algol68 !

When the command is obeyed, each new value is enclosed in its own cartouche. The result of obeying the command is in the enclosing cartouche as shown below:

source algol68 !

Note that this command includes source as an unnamed value and algol68 as the name of a procedure, whose name has been replaced by its value, algol68, by the command interpreter.

The unnamed value source algol68 ! is the result of obeying the command.

ed_date_doc:

This edfile,held within the first edfile, is the documentation on the filed procedure, Filed (Edfile->Vec Char). This procedure takes an edfile and delivers its creation date in the form dd/mm/yy hh mm. It was created from ed_date :Module .

Edfile Filed (Edfile->Vec Char) !
is a command delivering the date on which Edfile was created.

Note the convenience of holding documentation about the procedure with its value and its derivation all in one edfile.

The example given above is a tree structure. In general,however an edfile is not strictly speaking tree structured because the same value may appear in more than one place in the acyclic structure. I loosely refer to such a structure as a tree because it can conveniently be thought of as tree-like.

This example shows the use of unnamed values both in commands and in the structured edfile. A capability for the same value can be held in several places in the structured filestore, and each may be displayed with a different label in the cartouche. Only a single copy of any value exists, regardless of the number of copies of the capability for it or the number of different labels appearing in the cartouches that represent it. The label cannot be used to find the value represented by the cartouche because labels are not included in any dictionary associating names with values. The labels are used only to display information about that instance of the value. The default label in a cartouche gives the mode of the value that it represents.

Note that because filestore is non-overwriting, there is no need to take a back-up copy of any edfile or simple value to guard against unwanted changes; filestore objects cannot change underneath you. An apparent change to any filestore value involves getting a capability for a new (changed) value. The old (unchanged) value continues to exist as long as any instance of the capability for it is retained. When an editable file is apparently changed by editing, what actually happens is that the editor writes the edited file to a new filestore block and returns a new capability for the edited file. At this stage both the old and the new (changed) files exist as separate values so a user may have capabilities for both. He may choose to retain both capabilities or to discard his copy of the capability that he no longer wants. The old file has not changed and holders of a capability for it will still be able to access it. The user has simply gained access to another editable file.

The immutability of a filestore value contrasts with the use of names for values. On any system (including Flex) the same name may refer to different values at different times because the name/value association has been changed to associate a new value with the name. On systems other than Flex when a value is changed (updated) it may automatically continue to be associated with the same name following a convention which associates the most recent version of a value with its name. Even when a name/value association has not explicitly been changed the name may refer to different values at different times.

3.5 Identifying values

Because values can be held anonymously in the structured edfiles and can be used anonymously by the command interpreter, names on Flex do not have the importance that they assume on most other PSEs.

Each Flex user has a private dictionary of name/value associations for filestore values named by him. In practice these dictionaries tend to be very small. Only those values to which a user wishes to refer in a wide variety of different contexts are named, plus one outermost edfile that contains a structure of other edfiles, text, and values of various different Flex modes. All other objects are accessed (using the editor) via their position in some edfile. Only the outermost edfile need be named.

Typically there will be one named edfile for each major piece of work of interest to a user. The grouping of objects within a single structured edfile is purely a matter of convenience, a way of keeping related things together. In particular, documentation can be naturally structured with each chapter in a different sub-edfile, which in turn contains sub-edfiles for sections.

Systems that use tree-structured directories of name/value associations often use a structured naming convention to navigate from the root directory to any filestore value. The Flex editor, in addition to the usual functions of text file editing, allows the user to navigate through the tree-like structure to find unnamed values. After the editor has been called on a named edfile the editor displays a window on the outermost level of that file. To proceed down the tree-like structure to the next level in the tree, a user calls the editor recursively on an edfile in the current level of the tree. He proceeds, calling the editor recursively, until he reaches the position in the tree-like structure containing the object he seeks. This is usually a very rapid process, because the window selected for display at each call of the editor includes the cursor at its position when that edfile was last used. (In fact the position when it was created, since edfiles are not modified, only re-created.)

Values surrounded by explanatory text are easier to identify correctly than if the user must rely only on names, however well structured the dictionary hierarchy and however clear the naming convention. It is therefore easier to find an object in a tree structured hierarchy that includes descriptive text, (particularly because capabilities can be repeated wherever appropriate and can be displayed with helpful labels) than to keep track of an ever increasing dictionary of artificial file names. It is of course difficult to locate an anonymous value if the hierarchy is not well thought out. However, since capabilities can be repeated, it is possible to have an object (anonymous or named) in more than one hierarchy. Using the Flex editor it is easy to move the capability to a different place, or to restructure the hierarchy, so a badly structured hierarchy need not be a persistent problem.

3.6 The current name-space

Although values are frequently used anonymously, the Flex command interpreter can take a name and find an associated value. The meaning of a name in Flex is dependent on the context.

Name/value associations on Flex may be temporary or persistent. Persistent names persist from session to session and therefore apply only to filestore values. A user normally has access to two dictionaries of persistent names (his private dictionary and the common dictionary) although when users are created they can have any number of dictionaries associated with them. Dictionaries can be shared between individuals.

Temporary names do not persist between sessions and may therefore be associated with values of any Flex mode, including mainstore values. Temporary names are therefore useful for objects needed frequently during a session, and which contain mainstore capabilities, such as a vector of objects for testing a procedure. Although temporary name/value pairs are held in a temporary user dictionary in mainstore, temporary name/value associations can also be set up within any procedure, to apply only to calls to curt made within that procedure. This enables a user to write procedures to interact with the rest of the Flex PSE while using their own name space, that does not apply outside the procedures.

Flex dictionaries are not tree structured. They are usually so small that a tree structure is unnecessary. Tree structured dictionaries suffer from the disadvantage that a name is only in scope at the appropriate level in the tree, so it is necessary to navigate through a tree structure to the appropriate level for a name to have the correct meaning. If the same name can appear at different levels in a tree this can cause confusion. All names in a Flex dictionary are in scope simultaneously. There is less probability of confusion when using values directly from a structured edfile because the process of navigation throug' the edfile does not affect the name-space within which a user is w ing.

3.7 Module values and names

The unit of separate compilation on Flex is a value of Flex mode Module which like other values is usually stored and used anonymously. A module is a filestore value that gives access to the compiled code resulting from a single run of the compiler, to its external specification (the data and procedure types and source language names that are visible to users of the compiled unit) and to the source text from which it was derived (with certain restrictions to protect the source text from unauthorised access). A module also gives access indirectly to all the modules that it uses. The Algol68 source text of a module that uses other previously compiled modules includes an unnamed capability for each used module. This has the advantage of being context independent, whereas, if the reference to used modules were by name there would need to be a context (or library) for each module to associate the module names with the correct module or the correct version of a compiled unit.

A module on Flex is displayed by the editor in a cartouche with a label giving the source text name of the module. It is therefore easy when reading source text to identify the modules used by the unit being read. As elsewhere in Flex, the labels are not names: they are displayed for information.

A module value can be changed, so that the module capability gives access to a different compiled unit and associated values. (The change to the module does not violate the non-overwriting filestore. Modules are always accessed through a dictionary and the user is given a new dictionary, with the old module value replaced by the new one. The user need not be aware that the dictionary has changed. It is done automatically.) When a module value is changed after a recompilation, every instance of the module capability gives access to the new value. The effect of the change is immediate and system wide. (It is possible to test a new compiled unit before actually changing the module value.) Provided that the new value has the same external specification as the old value, the using modules need not be changed. They simply use the new value. If a change to a module involves a new external specification, each instance of the capability for the changed module is marked to show that the value must be revalidated before use. Capabilities for invalid module values are easily identified, and procedures exist to search for them and revalidate them, amending the using module to hold a capability for the revalidated module. This process is simplified by holding the values of the invalid modules in the using modules rather than their names because it cuts out the intermediate step of associating a name with the correct value.

Since used modules are included as context independent values rather than as names there is no need to change any build commands or module libraries to associate a name with the new version of the compiled unit. Build commands, that allow a user to rebuild a procedure or program after changes to some compiled units, are not required on Flex since executable images are not kept on Flex filestore. Procedure values are only assembled when they need to be loaded for execution. The current code from each used module is then included. Build commands are sometimes used on conventional systems to allow reconstruction of an old version of a program. The required compiled units are identified by name and version number or date. On Flex a procedure is provided that makes it possible to reconstruct an old version of a procedure by replacing selected modules used within the procedure. If the user has retained the old version of the source text or of the compiled unit he can use this to replace the current version. It is possible to arrange that all changes to modules are automatically recorded in a log file bound to the module. Other relevant information, such as the name and address of the author and source text of the superseded version may also be included. Version numbers and module names are not required for this process.

3.8 Shared values

Capabilities for those values that are shared by all Flex users are accessed through the common dictionary, which is accessible to every user. Most such values are not named. The capabilities are held in one common documentation edfile, named in the common dictionary. The common documentation file, containing capabilities not only for all shared documentation but also for modules, procedures and other shared values, is structured by topic. Capabilities are repeated as necessary if they belong to more than one topic. Users can extract the unnamed capabilities from the common documentation edfile for their own use. The shared values are not copied. Only the capabilities are copied.

3.9 Relationships

In most computer systems explicit relationships between different objects in the environment are provided only in databases. Usually these, if available, are provided in addition to the PSE rather than as an integral part of it. Naming conventions are employed on some systems to indicate implicit relationships between objects such as source text, compiled code and program and to associate a meaning with a value.

Names are not needed on Flex to indicate the meaning or the relationships of a value. Edfiles are database-like objects that implicitly express relationships by holding related values either adjacent or at different levels in the same edfile. For example, relating a set of text files such as a requirement specification, a design document, user guides etc. to the modules and procedures that implement them can be done by holding them all in the same edfile, together with the implementation, as illustrated in the edfile example, ed_date_doc. Meaning can be documented fully in the surrounding text rather than relying on the somewhat inadequate shorthand of appropriately chosen names. Multiple relationships may be expressed by holding a capability in more than one place.

The usual way to find a value on Flex filestore is to scan a structured edfile (provided the user knows roughly where in the structure to look). It is easy to write procedures to search from the outermost level of a container (such as an edfile, a dictionary or a module) for a value having some required characteristic. For example, a procedure could find all objects in an edfile of specified mode or a procedure could search for a module that uses a given module. It can be useful to discover which module keeps a specific item, such as an Algol68 mode, in its external specification and hence to look at the item in more detail. Using existing procedures from the common dictionary, it is easy to write such a procedure which, given the source text name of the sought item, searches for a module that keeps the item in its external specification, and delivers the module to the user.

There are currently some explicit relationships on Flex none of which relies on a naming convention. There is the binding relationship between any value and its mode and a relationship binding an edfile (including source text accessed from a module) to its creation date. Flex dictionaries express the relationship between a value, its documentation and the date of association between a name and the value. (A dictionary entry associates a name with both a value and an edfile holding its documentation. The information procedure on Flex delivers the documentation associated with the named value.) The module expresses the mandatory relationships between compiled code, its source text, its external specification and its log file. The module also expresses the relationship between a module and the modules that it uses. Other explicit relationships could be added to Flex using a data structure similar to mode Module in which related capabilities would be accessible through the same object.

4. Conclusions

I have described the use of structured filestore holding anonymous values, supported by a command language that uses capability values in preference to names. The advantages of such a system need to be experienced to be fully appreciated.

Capability values are context independent whereas names are context dependent. Indeed on some conventional systems the file associated with a name can be deleted while the name is still preserved (perhaps in a list of files needed to link a program). The value accessed by a capability cannot be deleted as long as the capability exists somewhere. Only when all capabilities for a value have been discarded is the value itself deleted.

Structured edfiles, that can hold values as well as text, make it possible to keep values on filestore without naming them. The value held in an edfile does not change with time because with non-overwriting filestore the edfile does not change. The use of unnamed values is encouraged by the recognition by the command language of unnamed values both for procedures and for their parameters. The use of values directly in the command language avoids the intermediate step of associating a value with a name and the potential ambiguity of context dependent names. Similarly, the use of values rather than names for modules eases the problems of propagating the results of changes to affected users by avoiding the intermediate step of associating a module name with a value in a specific library and of ensuring that the user knows which version of a module is being used.

Although several conventional systems support tree structured directories, more complex structures, with the same object appearing in more than one place in a tree-like structure, as in the Flex edfile, are less usual. It is easy to find an anonymous value in a tree structured edfile, with labelled values surrounded by explanatory text and associated values kept together. Repetition of the same capability in more than one hierarchy does not lead to multiple copies of the associated value and can be useful both for ease of access and to indicate relationships between values. The problem of changing name-space by moving to a different level in a tree structured directory is avoided by using very small single level dictionaries and keeping the structure in the edfiles.

Flex does not need to use naming conventions to show the type of an object. The Flex mode of any value can be displayed directly. Similarly Flex does not rely on naming conventions or on structured directories to indicate relationships. They are shown by keeping values in the same structured edfile or by using explicit relationships such as those provided between source text and compiled code and between modules.

The freedom from cumbersome naming conventions is a bonus. The programmer no longer has to invent names for every object he handles nor does he need to employ a complex naming convention or keep systematic records of what the names mean in different contexts. The difficulty of choosing new names for a wide variety of values is avoided and any problems arising either from associating a single value with different names in different contexts or from associating a single name with different values in different contexts, can be avoided.

The structured filestore just described is but one of several unusual and useful features of the Flex PSE. Future work on Flex is aimed at making the PSE and the ideas it demonstrates more widely available, and at improving the facilities. The underlying architecture is not expected to change, but additional facilities are being worked on to enable Flex to be networked and to support host/target software development. The possibility of implementing Flex with its powerful mode system on existing computer systems (without re-microcoding) is being considered as a topic for future research.

5. References

1. "Flex Firmware" by I.F.Currie, P.W.Edwards and J.M Foster. RSRE Report 81009. Sept 81.

2. "Flex: A working computer with an architecture based on procedure values." by I.F.Currie, P.W.Edwards and J.M Foster. RSRE Memorandum 3500. 1982.

3. "Curt: The command interpreter for Flex" by I.F.Currie and J.M.Foster. RSRE Memorandum 3522. 1983.

4. "Extending data typing beyond the bounds of programming languages" M.Stanley. RSRE Memorandum 3878. 1985.

5. "A personal evaluation of the Flex Programming Support Environment" M.Stanley. RSRE Memorandum in preparation, 1985

DOCUMENT CONTROL SHEET

Overall security classification of sheet ........UNCLASSIFIED........

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference MEMORANDUM 3901	3. Agency Reference	4. Report Security Classification U/C
5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location ROYAL SIGNALS AND RADAR ESTABLISHMENT		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		

7. Title
THE USE OF VALUES WITHOUT NAMES IN A PROGRAMMING SUPPORT ENVIRONMENT

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials STANLEY, M	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
11. Contract Number	12. Period	13. Project	14. Other Reference	

15. Distribution statement
UNLIMITED

Descriptors (or keywords)

continue on separate piece of paper

Abstract

The Flex PSE, developed at RSRE, Malvern, allows objects to be used without requiring that they be given names. This paper discusses how this is achieved, and the benefit to the user.

S80/48